目　录

1 排版

[1]1-1：程序块要采用缩进风格编写，缩进的空格数为4个。

说明：对于由开发工具自动生成的代码可以有不一致。

[1]1-2：相对独立的程序块之间、变量说明之后必须加空行。

示例：如下例子不符合规范。

```
if (!valid_ni(ni))
{
    ... // program code
}
repssn_ind = ssn_data[index].repssn_index;
repssn_ni  = ssn_data[index].ni;
```

应如下书写

```
if (!valid_ni(ni))
{
    ... // program code
}

repssn_ind = ssn_data[index].repssn_index;
repssn_ni  = ssn_data[index].ni;
```

[1]1-3：较长的语句（>80字符）要分成多行书写，长表达式要在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，使排版整齐，语句可读。

示例：

```
perm_count_msg.head.len = NO7_TO_STAT_PERM_COUNT_LEN
                      + STAT_SIZE_PER_FRAM * sizeof( _UL );

act_task_table[frame_id * STAT_TASK_CHECK_NUMBER + index].occupied
            = stat_poi[index].occupied;

act_task_table[taskno].duration_true_or_false
            = SYS_get_sccp_statistic_state( stat_item );

report_or_not_flag = ((taskno < MAX_ACT_TASK_NUMBER)
```

```
&& (n7stat_stat_item_valid (stat_item))
&& (act_task_table[taskno].result_data != 0));
```

[1]**1-4：循环、判断等语句中若有较长的表达式或语句，则要进行适应的划分，长表达式要在低优先级操作符处划分新行，操作符放在新行之首。**

示例：

```
if ((taskno < max_act_task_number)
    && (n7stat_stat_item_valid (stat_item)))
{
    ... // program code
}

for (i = 0, j = 0; (i < BufferKeyword[word_index].word_length)
                   && (j < NewKeyword.word_length); i++, j++)
{
    ... // program code
}

for (i = 0, j = 0;
     (i < first_word_length) && (j < second_word_length);
     i++, j++)
{
    ... // program code
}
```

[1]**1-5：若函数或过程中的参数较长，则要进行适当的划分。**

示例：

```
n7stat_str_compare((BYTE *) & stat_object,
                   (BYTE *) & (act_task_table[taskno].stat_object),
                   sizeof (_STAT_OBJECT));

n7stat_flash_act_duration( stat_item, frame_id *STAT_TASK_CHECK_NUMBER
                                      + index, stat_object );
```

[1]**1-6：不允许把多个短语句写在一行中，即一行只写一条语句。**

示例：如下例子不符合规范。

```
rect.length = 0;  rect.width = 0;
```

应如下书写

```
rect.length = 0;
rect.width  = 0;
```

¹**1-7：if、for、do、while、case、switch、default等语句自占一行，且if、for、do、while等语句的执行语句部分无论多少都要加括号{}。**

示例：如下例子不符合规范。

```
if (pUserCR == NULL) return;
```

应如下书写：

```
if (pUserCR == NULL)
{
    return;
}
```

¹**1-8：对齐只使用空格键，不使用TAB键。**

说明：以免用不同的编辑器阅读程序时，因 TAB 键所设置的空格数目不同而造成程序布局不整齐，不要使用 BC 作为编辑器合版本，因为 BC 会自动将 8 个空格变为一个 TAB 键，因此使用 BC 合入的版本大多会将缩进变乱。

¹**1-9：函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格，case语句下的情况处理语句也要遵从语句缩进要求。**

¹**1-10：程序块的分界符（如C/C++语言的大括号‘{’和‘}’）应各独占一行并且位于同一列，同时与引用它们的语句左对齐。在函数体的开始、类的定义、结构的定义、枚举的定义以及if、for、do、while、switch、case语句中的程序都要采用如上的缩进方式。**

示例：如下例子不符合规范。

```
for (...) {
    ... // program code
}

if (...)
    {
    ... // program code
    }
```



```
void example_fun( void )
    {
    ... // program code
    }
```

应如下书写。

```
for (...)
{
    ... // program code
}

if (...)
{
    ... // program code
}

void example_fun( void )
{
    ... // program code
}
```

¹1-11：在两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符（如－>），后不应加空格。

说明：采用这种松散方式编写代码的目的是使代码更加清晰。

由于留空格所产生的清晰性是相对的，所以，在已经非常清晰的语句中没有必要再留空格，如果语句已足够清晰则括号内侧(即左括号后面和右括号前面)不需要加空格，多重括号间不必加空格，因为在 C/C++语言中括号已经是最清晰的标志了。

在长语句中，如果需要加的空格非常多，那么应该保持整体清晰，而在局部不加空格。给操作符留空格时不要连续留两个以上空格。

示例：

(1) 逗号、分号只在后面加空格。

```
int a, b, c;
```

(2)比较操作符，赋值操作符"="、 "+="，算术操作符"+"、"%"，逻辑操作符"&&"、"&"，位域操作符"<<"、"^"等双目操作符的前后加空格。

```
if (current_time >= MAX_TIME_VALUE)
a = b + c;
a *= 2;
a = b ^ 2;
```

(3)"!"、"~"、"++"、"--"、"&"（地址运算符）等单目操作符前后不加空格。

```
*p = 'a';        // 内容操作"*"与内容之间
flag = !isEmpty; // 非操作"!"与内容之间
p = &mem;        // 地址操作"&" 与内容之间
i++;             // "++","--"与内容之间
```

(4)"->"、"."前后不加空格。

```
p->id = pid;     // "->"指针前后不加空格
```

(5) if、for、while、switch 等与后面的括号间应加空格，使 if 等关键字更为突出、明显。

```
if (a >= b && c > d)
```

½1-1：一行程序以小于80字符为宜，不要写得过长。

2 注释

[1]2-1：一般情况下，源程序有效注释量必须在20％以上。

说明：注释的原则是有助于对程序的阅读理解，在该加的地方都加了，注释不宜太多也不能太少，注释语言必须准确、易懂、简洁。

[1]2-2：说明性文件（如头文件.h文件、.inc文件、.def文件、编译说明文件.cfg等）头部应进行注释，注释必须列出：版权说明、版本号、生成日期、作者、内容、功能、与其它文件的关系、修改日志等，头文件的注释中还应有函数功能简要说明。

示例：下面这段头文件的头注释比较标准，当然，并不局限于此格式，但上述信息建议要包含在内。

```
/*************************************************
  Copyright (C), 1988-1999, Huawei Tech. Co., Ltd.
  File name:      // 文件名
  Author:      Version:       Date: // 作者、版本及完成日期
  Description:    // 用于详细说明此程序文件完成的主要功能，与其他模块
                  // 或函数的接口，输出值、取值范围、含义及参数间的控
                  // 制、顺序、独立或依赖等关系
  Others:         // 其它内容的说明
  Function List:  // 主要函数列表，每条记录应包括函数名及功能简要说明
    1. ....
  History:        // 修改历史记录列表，每条修改记录应包括修改日期、修改
                  // 者及修改内容简述
    1. Date:
       Author:
       Modification:
    2. ...
*************************************************/
```

[1]2-3：源文件头部应进行注释，列出：版权说明、版本号、生成日期、作者、模块目的/功能、主要函数及其功能、修改日志等。

示例：下面这段源文件的头注释比较标准，当然，并不局限于此格式，但上述信息建议要包含在内。

```
/*************************************************************
  Copyright (C), 1988-1999, Huawei Tech. Co., Ltd.
  FileName: test.cpp
  Author:       Version :        Date:
  Description:    // 模块描述
  Version:        // 版本信息
  Function List:   // 主要函数及其功能
    1. -------
  History:        // 历史修改记录
      <author>  <time>   <version >   <desc>
      David    96/10/12     1.0     build this moudle
***********************************************************/
```

说明：Description 一项描述本文件的内容、功能、内部各部分之间的关系及本文件与其它文件关系等。History 是修改历史记录列表，每条修改记录应包括修改日期、修改者及修改内容简述。

[1]**2-4：函数头部应进行注释，列出：函数的目的/功能、输入参数、输出参数、返回值、调用关系（函数、表）等。**

示例：下面这段函数的注释比较标准，当然，并不局限于此格式，但上述信息建议要包含在内。

```
/*************************************************
  Function:       // 函数名称
  Description:    // 函数功能、性能等的描述
  Calls:          // 被本函数调用的函数清单
  Called By:      // 调用本函数的函数清单
  Table Accessed: // 被访问的表（此项仅对于牵扯到数据库操作的程序）
  Table Updated:  // 被修改的表（此项仅对于牵扯到数据库操作的程序）
  Input:          // 输入参数说明，包括每个参数的作
                 // 用、取值说明及参数间关系。
  Output:         // 对输出参数的说明。
  Return:         // 函数返回值的说明
  Others:         // 其它说明
*************************************************/
```

[1]**2-5：边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性。不再有用的注释要删除。**

[1]**2-6：注释的内容要清楚、明了，含义准确，防止注释二义性。**

说明：错误的注释不但无益反而有害。

规则2-7：避免在注释中使用缩写，特别是非常用缩写。

说明：在使用缩写时或之前，应对缩写进行必要的说明。

[1]**2-8：注释应与其描述的代码相近，对代码的注释应放在其上方或右方（对单条语句的注释）相邻位置，不可放在下面，如放于上方则需与其上面的代码用空行隔开。**

示例：如下例子不符合规范。

例 1：

```
/* get replicate sub system index and net indicator */



repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

例 2：

```
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
/* get replicate sub system index and net indicator */
```

应如下书写

```
/* get replicate sub system index and net indicator */
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

[1]**2-9：对于所有有物理含义的变量、常量，如果其命名不是充分自注释的，在声明时都必须加以注释，说明其物理含义。变量、常量、宏的注释应放在其上方相邻位置或右方。**

示例：

```
/* active statistic task number */
#define MAX_ACT_TASK_NUMBER 1000
```

```
#define MAX_ACT_TASK_NUMBER 1000 /* active statistic task number */
```

¹2-10：数据结构声明(包括数组、结构、类、枚举等)，如果其命名不是充分自注释的，必须加以注释。对数据结构的注释应放在其上方相邻位置，不可放在下面；对结构中的每个域的注释放在此域的右方。

示例：可按如下形式说明枚举/数据/联合结构。

```
/* sccp interface with sccp user primitive message name */
enum  SCCP_USER_PRIMITIVE
{
    N_UNITDATA_IND, /* sccp notify sccp user unit data come */
    N_NOTICE_IND,   /* sccp notify user the No.7 network can not */
                    /* transmission this message */
    N_UNITDATA_REQ, /* sccp user's unit data transmission request*/
};
```

¹2-11：全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取它以及存取时注意事项等的说明。

示例：

```
/* The ErrorCode when SCCP translate */
/* Global Title failure, as follows */      // 变量作用、含义
/* 0 — SUCCESS   1 — GT Table error */
/* 2 — GT error  Others — no use  */       // 变量取值范围
/* only  function  SCCPTranslate() in */
/* this modual can modify it,  and  other */
/* module can visit it through call */
/* the  function GetGTTransErrorCode() */    // 使用方法
BYTE g_GTTranErrorCode;
```

¹2-12：注释与所描述内容进行同样的缩排。

说明：可使程序排版整齐，并方便注释的阅读与理解。

示例：如下例子，排版不整齐，阅读稍感不方便。

```
void example_fun( void )
{
/* code one comments */
    CodeBlock One
```

```
        /* code two comments */
    CodeBlock Two
}
```

应改为如下布局。

```
void example_fun( void )
{
    /* code one comments */
    CodeBlock One

    /* code two comments */
    CodeBlock Two
}
```

[1]**2-13：将注释与其上面的代码用空行隔开。**

示例：如下例子，显得代码过于紧凑。

```
/* code one comments */
program code one
/* code two comments */
program code two
```

应如下书写

```
/* code one comments */
program code one

/* code two comments */
program code two
```

[1]**2-14：对变量的定义和分支语句（条件分支、循环语句等）必须编写注释。**

说明：这些语句往往是程序实现某一特定功能的关键，对于维护人员来说，良好的注释帮助更好的理解程序，有时甚至优于看设计文档。

[1]**2-15：对于switch语句下的case语句，如果因为特殊情况需要处理完一个case后进入下一个case处理，必须在该case语句处理完、下一个case语句前加上明确的注释。**

说明：这样比较清楚程序编写者的意图，有效防止无故遗漏 break 语句。

示例（注意斜体加粗部分）：

```
case CMD_UP:
    ProcessUp();
    break;

case CMD_DOWN:
    ProcessDown();
    break;

case CMD_FWD:
    ProcessFwd();

if (...)
{
    ...
    break;
}
else
{
    ProcessCFW_B();   // now jump into case CMD_A
}

case CMD_A:
    ProcessA();
    break;

case CMD_B:
    ProcessB();
    break;

case CMD_C:
    ProcessC();
     break;

case CMD_D:
    ProcessD();
```

```
    break;
...
```

½2-1：避免在一行代码或表达式的中间插入注释。

说明：除非必要，不应在代码或表达中间插入注释，否则容易使代码可理解性变差。

½2-2：通过对函数或过程、变量、结构等正确的命名以及合理地组织代码的结构，使代码成为自注释的。

说明：清晰准确的函数、变量等的命名，可增加代码可读性，并减少不必要的注释。

½2-3：在代码的功能、意图层次上进行注释，提供有用、额外的信息。

说明：注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，帮助读者理解代码，防止没必要的重复注释信息。

示例：如下注释意义不大。

```
/* if receive_flag is TRUE */
if (receive_flag)
```

而如下的注释则给出了额外有用的信息。

```
/* if mtp receive a message from links */
if (receive_flag)
```

½2-4：在程序块的结束行右方加注释标记，以表明某程序块的结束。

说明：当代码段较长，特别是多重嵌套时，这样做可以使代码更清晰，更便于阅读。

示例：参见如下例子。

```
if (...)
{
    // program code

    while (index < MAX_INDEX)
    {
        // program code
    } /* end of while (index < MAX_INDEX) */ // 指明该条while语句结束
} /* end of  if (...)*/ // 指明是哪条if语句结束
```

½2-5：注释格式尽量统一，建议使用“/* …… */”。

½2-6：注释应考虑程序易读及外观排版的因素，使用的语言若是中、英兼有的，建议多使用中文，除非能用非常流利准确的英文表达。

说明：注释语言不统一，影响程序易读性和外观排版，出于对维护人员的考虑，建议使用中文。

3 标识符命名

¹3-1：标识符的命名要清晰、明了，有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解。

说明：较短的单词可通过去掉“元音”形成缩写；较长的单词可取单词的头几个字母形成缩写；一些单词有大家公认的缩写。

示例：如下单词的缩写能够被大家基本认可。

temp 可缩写为 tmp ;

flag 可缩写为 flg ;

statistic 可缩写为 stat ;

increment 可缩写为 inc ;

message 可缩写为 msg ;

¹3-2：命名中若使用特殊约定或缩写，则要有注释说明。

说明：应该在源文件的开始之处，对文件中所使用的缩写或约定，特别是特殊的缩写，进行必要的注释说明。

¹3-3：自己特有的命名风格，要自始至终保持一致，不可来回变化。

说明：个人的命名风格，在符合所在项目组或产品组的命名规则的前提下，才可使用。(即命名规则中没有规定到的地方才可有个人命名风格)。

¹3-4：对于变量命名，禁止取单个字符（如i、j、k...），建议除了要有具体含义外，还能表明其变量类型、数据类型等，但i、j、k作局部循环变量是允许的。

说明：变量，尤其是局部变量，如果用单个字符表示，很容易敲错（如 i 写成 j），而编译时又检查不出来，有可能为了这个小小的错误而花费大量的查错时间。

示例：下面所示的局部变量名的定义方法可以借鉴。

int liv_Width

其变量名解释如下：

l　局部变量（Local）　（其它：g　全局变量（Global）...）

i　数据类型（Interger）

v　变量（Variable）　（其它：c　常量（Const）...）

Width　变量含义

这样可以防止局部变量与全局变量重名。

¹3-5：命名规范必须与所使用的系统风格保持一致，并在同一项目中统一，比如采用UNIX的全小写加下划线的风格或大小写混排的方式，不要使用大小写与下划线混排的方式，用作特殊标识如标识成员变量或全局变量的m_和g_，其后加上大小写混排的方式是允许的。

示例： Add_User 不允许，add_user、AddUser、m_AddUser 允许。

½3-1：除非必要，不要用数字或较奇怪的字符来定义标识符。

示例：如下命名，使人产生疑惑。

```
#define _EXAMPLE_0_TEST_
#define _EXAMPLE_1_TEST_
void set_sls00( BYTE sls );
```

应改为有意义的单词命名

```
#define _EXAMPLE_UNIT_TEST_
#define _EXAMPLE_ASSERT_TEST_
void set_udt_msg_sls( BYTE sls );
```

½3-2：在同一软件产品内，应规划好接口部分标识符（变量、结构、函数及常量）的命名，防止编译、链接时产生冲突。

说明：对接口部分的标识符应该有更严格限制，防止冲突。如可规定接口部分的变量与常量之前加上“ 模块 ”标识等。

½3-3：用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。

说明：下面是一些在软件中常用的反义词组。

```
add / remove      begin / end       create / destroy
insert / delete    first / last      get / release
increment / decrement              put / get
add / delete      lock / unlock     open / close
min / max        old / new         start / stop
next / previous    source / target    show / hide
send / receive    source / destination
cut / paste       up / down
```

示例：

```
int  min_sum;
```

```
int  max_sum;
int  add_user( BYTE *user_name );
int  delete_user( BYTE *user_name );
```

½3-4：除了编译开关/头文件等特殊应用，应避免使用_EXAMPLE_TEST_之类以下划线开始和结尾的定义。

4 可读性

¹4-1：注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认优先级。

说明：防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错。

示例：下列语句中的表达式

```
word = (high << 8) | low    (1)
if ((a | b) && (a & c))     (2)
if ((a | b) < (c & d))      (3)
```

如果书写为

```
high << 8 | low
a | b && a & c
a | b < c & d
```

由于

```
high << 8 | low = ( high << 8) | low,
a | b && a & c = (a | b) && (a & c),
```

(1)(2)不会出错，但语句不易理解；

a | b < c & d = a | （b < c） & d，(3)造成了判断条件出错。

¹4-2：避免使用不易理解的数字，用有意义的标识来替代。涉及物理状态或者含有物理意义的常量，不应直接使用数字，必须用有意义的枚举或宏来代替。

示例：如下的程序可读性差。

```
if (Trunk[index].trunk_state == 0)
{
   Trunk[index].trunk_state = 1;
   ...  // program code
}
```

应改为如下形式。

```
#define TRUNK_IDLE 0
#define TRUNK_BUSY 1

if (Trunk[index].trunk_state == TRUNK_IDLE)
{
   Trunk[index].trunk_state = TRUNK_BUSY;
```

```
    ...  // program code
}
```

½4-1：源程序中关系较为紧密的代码应尽可能相邻。

说明：便于程序阅读和查找。

示例：以下代码布局不太合理。

```
rect.length = 10;
char_poi = str;
rect.width = 5;
```

若按如下形式书写，可能更清晰一些。

```
rect.length = 10;
rect.width = 5; // 矩形的长与宽关系较密切，放在一起。
char_poi = str;
```

½4-2：不要使用难懂的技巧性很高的语句，除非很有必要时。

说明：高技巧语句不等于高效率的程序，实际上程序的效率关键在于算法。

示例：如下表达式，考虑不周就可能出问题，也较难理解。

```
* stat_poi ++ += 1;

* ++ stat_poi += 1;
```

应分别改为如下。

```
*stat_poi += 1;
stat_poi++;     // 此二语句功能相当于“ * stat_poi ++ += 1; ”

++ stat_poi;
*stat_poi += 1; // 此二语句功能相当于“ * ++ stat_poi += 1; ”
```

5 变量、结构

[1]5-1：去掉没必要的公共变量。

说明：公共变量是增大模块间耦合的原因之一，故应减少没必要的公共变量以降低模块间的耦合度。

[1]5-2：仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系。

说明：在对变量声明的同时，应对其含义、作用及取值范围进行注释说明，同时若有必要还应说明与其它变量的关系。

[1]5-3：明确公共变量与操作此公共变量的函数或过程的关系，如访问、修改及创建等。

说明：明确过程操作变量的关系后，将有利于程序的进一步优化、单元测试、系统联调以及代码维护等。这种关系的说明可在注释或文档中描述。

示例：在源文件中，可按如下注释形式说明。

```
RELATION    System_Init    Input_Rec    Print_Rec    Stat_Score
Student     Create         Modify       Access       Access
Score       Create         Modify       Access       Access, Modify
```

注：RELATION 为操作关系；System_Init、Input_Rec、Print_Rec、Stat_Score 为四个不同的函数；Student、Score 为两个全局变量；Create 表示创建，Modify 表示修改，Access 表示访问。

其中，函数 Input_Rec、Stat_Score 都可修改变量 Score，故此变量将引起函数间较大的耦合，并可能增加代码测试、维护的难度。

[1]5-4：当向公共变量传递数据时，要十分小心，防止赋与不合理的值或越界等现象发生。

说明：对公共变量赋值时，若有必要应进行合法性检查，以提高代码的可靠性、稳定性。

[1]5-5：防止局部变量与公共变量同名。

说明：若使用了较好的命名规则，那么此问题可自动消除。

[1]5-6：严禁使用未经初始化的变量作为右值。

说明：特别是在 C/C++中引用未经赋值的指针，经常会引起系统崩溃。

½5-1：构造仅有一个模块或函数可以修改、创建，而其余有关模块或函数只访问的公共变量，防止多个不同模块或函数都可以修改、创建同一公共变量的现象。

说明：降低公共变量耦合度。

½5-2：使用严格形式定义的、可移植的数据类型，尽量不要使用与具体硬件或软件环境关系密切的变量。

说明：使用标准的数据类型，有利于程序的移植。

示例：如下例子（在 DOS 下 BC3.1 环境中），在移植时可能产生问题。

```
void main()
{
    register int index; // 寄存器变量

    _AX = 0x4000; // _AX 是 BC3.1 提供的寄存器“伪变量”
    ... // program code
}
```

½5-3：结构的功能要单一，是针对一种事务的抽象。

说明：设计结构时应力争使结构代表一种现实事务的抽象，而不是同时代表多种。结构中的各元素应代表同一事务的不同侧面，而不应把描述没有关系或关系很弱的不同事务的元素放到同一结构中。

示例：如下结构不太清晰、合理。

```
typedef struct STUDENT_STRU
{
    unsigned char name[8]; /* student's name */
    unsigned char age;     /* student's age */
    unsigned char sex;     /* student's sex, as follows */
                           /* 0 - FEMALE; 1 - MALE */
    unsigned char
           teacher_name[8]; /* the student teacher's name */
    unisgned char
           teacher_sex;     /* his teacher sex */
} STUDENT;
```

若改为如下，可能更合理些。

```
typedef struct TEACHER_STRU
{
    unsigned char name[8]; /* teacher name */
    unisgned char sex;     /* teacher sex, as follows */
                           /* 0 - FEMALE; 1 - MALE */
} TEACHER;

typedef struct STUDENT_STRU
{
    unsigned char name[8];     /* student's name */
    unsigned char age;         /* student's age */
    unsigned char sex;         /* student's sex, as follows */
                               /* 0 - FEMALE; 1 - MALE */
    unsigned int  teacher_ind; /* his teacher index */
} STUDENT;
```

½5-4：不要设计面面俱到、非常灵活的数据结构。

说明：面面俱到、灵活的数据结构反而容易引起误解和操作困难。

½5-5：不同结构间的关系不要过于复杂。

说明：若两个结构间关系较复杂、密切，那么应合为一个结构。

示例：如下两个结构的构造不合理。

```
typedef struct PERSON_ONE_STRU
{
    unsigned char name[8];
    unsigned char addr[40];
    unsigned char sex;
    unsigned char city[15];
} PERSON_ONE;

typedef struct PERSON_TWO_STRU
{
    unsigned char name[8];
    unsigned char age;
    unsigned char tel;
} PERSON_TWO;
```

由于两个结构都是描述同一事物的，那么不如合成一个结构。

```
typedef struct PERSON_STRU
{
    unsigned char name[8];
    unsigned char age;
    unsigned char sex;
    unsigned char addr[40];
    unsigned char city[15];
    unsigned char tel;
} PERSON;
```

½5-6：结构中元素的个数应适中。若结构中元素个数过多可考虑依据某种原则把元素组成不同的子结构，以减少原结构中元素的个数。

说明：增加结构的可理解性、可操作性和可维护性。

示例：假如认为如上的_PERSON 结构元素过多，那么可如下对之划分。

```
typedef struct PERSON_BASE_INFO_STRU
{
    unsigned char name[8];
    unsigned char age;
    unsigned char sex;
} PERSON_BASE_INFO;

typedef struct PERSON_ADDRESS_STRU
{
    unsigned char addr[40];
    unsigned char city[15];
    unsigned char tel;
} PERSON_ADDRESS;

typedef struct PERSON_STRU
{
    PERSON_BASE_INFO person_base;
    PERSON_ADDRESS person_addr;
} PERSON;
```

½5-7：仔细设计结构中元素的布局与排列顺序，使结构容易理解、节省占用空间，并减少引起误用现象。

说明：合理排列结构中元素顺序，可节省空间并增加可理解性。

示例：如下结构中的位域排列，将占较大空间，可读性也稍差。

```
typedef struct EXAMPLE_STRU
{
    unsigned int valid: 1;
    PERSON person;
    unsigned int set_flg: 1;
} EXAMPLE;
```

若改成如下形式，不仅可节省 1 字节空间，可读性也变好了。

```
typedef struct EXAMPLE_STRU
{
    unsigned int valid: 1;
    unsigned int set_flg: 1;
    PERSON person ;
} EXAMPLE;
```

½5-8：结构的设计要尽量考虑向前兼容和以后的版本升级，并为某些未来可能的应用保留余地（如预留一些空间等）。

说明：软件向前兼容的特性，是软件产品是否成功的重要标志之一。如果要想使产品具有较好的前向兼容，那么在产品设计之初就应为以后版本升级保留一定余地，并且在产品升级时必须考虑前一版本的各种特性。

½5-9：留心具体语言及编译器处理不同数据类型的原则及有关细节。

说明：如在 C 语言中，static 局部变量将在内存“数据区”中生成，而非 static 局部变量将在“堆栈”中生成。这些细节对程序质量的保证非常重要。

½5-10：编程时，要注意数据类型的强制转换。

说明：当进行数据类型强制转换时，其数据的意义、转换后的取值等都有可能发生变化，而这些细节若考虑不周，就很有可能留下隐患。

½5-11：对编译系统默认的数据类型转换，也要有充分的认识。

示例：如下赋值，多数编译器不产生告警，但值的含义还是稍有变化。

```
char chr;
unsigned short int exam;

chr = -1;
exam = chr; // 编译器不产生告警，此时exam为0xFFFF。
```

½5-12：尽量减少没有必要的数据类型默认转换与强制转换。

½5-13：合理地设计数据并使用自定义数据类型，避免数据间进行不必要的类型转换。

½5-14：对自定义数据类型进行恰当命名，使它成为自描述性的，以提高代码可读性。注意其命名方式在同一产品中的统一。

说明：使用自定义类型，可以弥补编程语言提供类型少、信息量不足的缺点，并能使程序清晰、简洁。

示例：可参考如下方式声明自定义数据类型。

下面的声明可使数据类型的使用简洁、明了。

```
typedef unsigned char  BYTE;
typedef unsigned short WORD;
typedef unsigned int   DWORD;
```

下面的声明可使数据类型具有更丰富的含义。

```
typedef float DISTANCE;
typedef float SCORE;
```

½5-15：当声明用于分布式环境或不同CPU间通信环境的数据结构时，必须考虑机器的字节顺序、使用的位域及字节对齐等问题 。

说明：比如Intel CPU与68360 CPU，在处理位域及整数时，其在内存存放的“顺序”正好相反。

示例：假如有如下短整数及结构。

```
unsigned short int exam;
typedef struct EXAM_BIT_STRU
{                       /* Intel 68360 */
    unsigned int A1: 1; /* bit  0     7   */
    unsigned int A2: 1; /* bit  1     6   */
```

```
   unsigned int A3: 1; /* bit  2     5   */
} EXAM_BIT;
```

如下是 Intel CPU 生成短整数及位域的方式。

内存： 0   1   2   ...   （从低到高，以字节为单位）

exam exam 低字节 exam 高字节

内存：   0 bit   1 bit   2 bit   ...   （字节的各“位”）

EXAM_BIT   A1   A2   A3

如下是 68360 CPU 生成短整数及位域的方式。

内存： 0   1   2   ...   （从低到高，以字节为单位）

exam exam 高字节 exam 低字节

内存：   7 bit   6 bit   5 bit   ...   （字节的各“位”）

EXAM_BIT   A1   A2   A3

说明：在对齐方式下，CPU 的运行效率要快得多。

示例：如下图，当一个 long 型数（如图中 long1）在内存中的位置正好与内存的字边界对齐时，CPU 存取这个数只需访问一次内存，而当一个 long 型数（如图中的 long2）在内存中的位置跨越了字边界时，CPU 存取这个数就需要多次访问内存，如 i960cx 访问这样的数需读内存三次（一个 BYTE、一个 SHORT、一个 BYTE，由 CPU 的微代码执行，对软件透明），所有对齐方式下 CPU 的运行效率明显快多了。

```
1       8       16      24      32
------- ------- ------- -------
| long1 | long1 | long1 | long1 |
------- ------- ------- -------
|       |       |       | long2 |
------- ------- ------- --------
| long2 | long2 | long2 |       |
------- ------- ------- --------
| ....
```

6 函数、过程

[1]6-1：对所调用函数的错误返回码要仔细、全面地处理。

[1]6-2：明确函数功能，精确（而不是近似）地实现函数设计。

[1]6-3：编写可重入函数时，应注意局部变量的使用（如编写C/C++语言的可重入函数时，应使用auto即缺省态局部变量或寄存器变量）。

说明：编写 C/C++语言的可重入函数时，不应使用 static 局部变量，否则必须经过特殊处理，才能使函数具有可重入性。

[1]6-4：编写可重入函数时，若使用全局变量，则应通过关中断、信号量（即P、V操作）等手段对其加以保护。

说明：若对所使用的全局变量不加以保护，则此函数就不具有可重入性，即当多个进程调用此函数时，很有可能使有关全局变量变为不可知状态。

示例：假设 Exam 是 int 型全局变量，函数 Squre_Exam 返回 Exam 平方值。那么如下函数不具有可重入性。

```
unsigned int example( int para )
{
    unsigned int temp;

    Exam = para; //  (**)
    temp = Square_Exam( );

    return temp;
}
```

此函数若被多个进程调用的话，其结果可能是未知的，因为当（**）语句刚执行完后，另外一个使用本函数的进程可能正好被激活，那么当新激活的进程执行到此函数时，将使 Exam 赋与另一个不同的 para 值，所以当控制重新回到“temp = Square_Exam( )”后，计算出的 temp 很可能不是预想中的结果。此函数应如下改进。

```
unsigned int example( int para )
{
    unsigned int temp;
```

```
    [申请信号量操作]          // 若申请不到“信号量”，说明另外的进程正处于
    Exam = para;              // 给 Exam 赋值并计算其平方过程中（即正在使用此
    temp = Square_Exam( );   // 信号），本进程必须等待其释放信号后，才可继
    [释放信号量操作]          // 续执行。若申请到信号，则可继续执行，但其
                            // 它进程必须等待本进程释放信号量后，才能再使
                            // 用本信号。
    return temp;
}
```

¹6-5：在同一项目组应明确规定对接口函数参数的合法性检查应由函数的调用者负责还是由接口函数本身负责，缺省是由函数调用者负责。

说明：对于模块间接口函数的参数的合法性检查这一问题，往往有两个极端现象，即：要么是调用者和被调用者对参数均不作合法性检查，结果就遗漏了合法性检查这一必要的处理过程，造成问题隐患；要么就是调用者和被调用者均对参数进行合法性检查，这种情况虽不会造成问题，但产生了冗余代码，降低了效率。

½6-1：防止将函数的参数作为工作变量。

说明：将函数的参数作为工作变量，有可能错误地改变参数内容，所以很危险。对必须改变的参数，最好先用局部变量代之，最后再将该局部变量的内容赋给该参数。

示例：下函数的实现不太好。

```
void sum_data( unsigned int num, int *data, int *sum )
{
    unsigned int count;

    *sum = 0;
    for (count = 0; count < num; count++)
    {
        *sum  += data[count]; // sum 成了工作变量，不太好。
    }
}
```

若改为如下，则更好些。

```
void sum_data( unsigned int num, int *data, int *sum )
```

```
{
    unsigned int count ;
    int sum_temp;

    sum_temp = 0;
    for (count = 0; count < num; count ++)
    {
        sum_temp  += data[count];
    }

    *sum = sum_temp;
}
```

½6-2：函数的规模尽量限制在200行以内。

说明：不包括注释和空格行。

½6-3：一个函数仅完成一件功能。

½6-4：为简单功能编写函数。

说明：虽然为仅用一两行就可完成的功能去编函数好象没有必要，但用函数可使功能明确化，增加程序可读性，亦可方便维护、测试。

示例：如下语句的功能不很明显。

```
value = ( a > b ) ? a : b ;
```

改为如下就很清晰了。

```
int max (int a, int b)
{
    return ((a > b) ? a : b);
}

value = max (a, b);
```

或改为如下。

```
#define MAX (a, b) (((a) > (b)) ? (a) : (b))

value = MAX (a, b);
```

½6-5：不要设计多用途面面俱到的函数。

说明：多功能集于一身的函数，很可能使函数的理解、测试、维护等变得困难。

½6-6：函数的功能应该是可以预测的，也就是只要输入数据相同就应产生同样的输出。

说明：带有内部“存储器”的函数的功能可能是不可预测的，因为它的输出可能取决于内部存储器（如某标记）的状态。这样的函数既不易于理解又不利于测试和维护。在 C/C++ 语言中，函数的 static 局部变量是函数的内部存储器，有可能使函数的功能不可预测，然而，当某函数的返回值为指针类型时，则必须是 STATIC 的局部变量的地址作为返回值，若为 AUTO 类，则返回为错针。

示例：如下函数，其返回值（即功能）是不可预测的。

```
unsigned int integer_sum( unsigned int base )
{
    unsigned int index;
    static unsigned int sum = 0; // 注意，是 static 类型的。
                                 // 若改为 auto 类型，则函数即变为可预测。
    for (index = 1; index <= base; index++)
    {
        sum += index;
    }

    return sum;
}
```

½6-7：尽量不要编写依赖于其他函数内部实现的函数。

说明：此条为函数独立性的基本要求。由于目前大部分高级语言都是结构化的，所以通过具体语言的语法要求与编译器功能，基本就可以防止这种情况发生。但在汇编语言中，由于其灵活性，很可能使函数出现这种情况。

示例：如下是在 DOS 下 TASM 的汇编程序例子。过程 Print_Msg 的实现依赖于 Input_Msg 的具体实现，这种程序是非结构化的，难以维护、修改。

```
...  // 程序代码
proc Print_Msg // 过程（函数）Print_Msg
    ...  // 程序代码
    jmp  LABEL
    ...  // 程序代码
```

```
endp

proc Input_Msg // 过程（函数）Input_Msg
    ...  // 程序代码
LABEL:
    ...  // 程序代码
endp
```

½6-8：避免设计多参数函数，不使用的参数从接口中去掉。

说明：目的减少函数间接口的复杂度。

½6-9：非调度函数应减少或防止控制参数，尽量只使用数据参数。

说明：本建议目的是防止函数间的控制耦合。调度函数是指根据输入的消息类型或控制命令，来启动相应的功能实体（即函数或过程），而本身并不完成具体功能。控制参数是指改变函数功能行为的参数，即函数要根据此参数来决定具体怎样工作。非调度函数的控制参数增加了函数间的控制耦合，很可能使函数间的耦合度增大，并使函数的功能不唯一。

示例：如下函数构造不太合理。

```
int add_sub( int a, int b, unsigned char add_sub_flg )
{
    if (add_sub_flg == INTEGER_ADD)
    {
        return (a + b);
    }
    else
    {
        return (a   b);
    }
}
```

不如分为如下两个函数清晰。

```
int add( int a, int b )
{
    return (a + b);
}
```

```
int sub( int a, int b )
{
   return (a   b);
}
```

½6-10：检查函数所有参数输入的有效性。

½6-11：检查函数所有非参数输入的有效性，如数据文件、公共变量等。

说明：函数的输入主要有两种：一种是参数输入；另一种是全局变量、数据文件的输入，即非参数输入。函数在使用输入之前，应进行必要的检查。

½6-12：函数名应准确描述函数的功能。

½6-13：使用动宾词组为执行某操作的函数命名。如果是OOP方法，可以只有动词（名词是对象本身）。

示例：参照如下方式命名函数。

```
void print_record( unsigned int rec_ind ) ;
int  input_record( void ) ;
unsigned char get_current_color( void ) ;
```

建议6-14：避免使用无意义或含义不清的动词为函数命名。

说明：避免用含义不清的动词如 process、handle 等为函数命名，因为这些动词并没有说明要具体做什么。

建议6-15：函数的返回值要清楚、明了，让使用者不容易忽视错误情况。

说明：函数的每种出错返回值的意义要清晰、明了、准确，防止使用者误用、理解错误或忽视错误返回码。

½6-16：除非必要，最好不要把与函数返回值类型不同的变量，以编译系统默认的转换方式或强制的转换方式作为返回值返回。

½6-17：让函数在调用点显得易懂、容易理解。

½6-18：在调用函数填写参数时，应尽量减少没有必要的默认数据类型转换或强制数据类型转换。

说明：因为数据类型转换或多或少存在危险。

½6-19：避免函数中不必要语句，防止程序中的垃圾代码。

说明：程序中的垃圾代码不仅占用额外的空间，而且还常常影响程序的功能与性能，很可能给程序的测试、维护等造成不必要的麻烦。

½6-20：防止把没有关联的语句放到一个函数中。

说明：防止函数或过程内出现随机内聚。随机内聚是指将没有关联或关联很弱的语句放到同一个函数或过程中。随机内聚给函数或过程的维护、测试及以后的升级等造成了不便，同时也使函数或过程的功能不明确。使用随机内聚函数，常常容易出现在一种应用场合需要改进此函数，而另一种应用场合又不允许这种改进，从而陷入困境。

在编程时，经常遇到在不同函数中使用相同的代码，许多开发人员都愿把这些代码提出来，并构成一个新函数。若这些代码关联较大并且是完成一个功能的，那么这种构造是合理的，否则这种构造将产生随机内聚的函数。

示例：如下函数就是一种随机内聚。

```
void Init_Var( void )
{
    Rect.length = 0;
    Rect.width = 0; /* 初始化矩形的长与宽 */

    Point.x = 10;
    Point.y = 10;   /* 初始化“点”的坐标 */
}
```

矩形的长、宽与点的坐标基本没有任何关系，故以上函数是随机内聚。

应如下分为两个函数：

```
void Init_Rect( void )
{
    Rect.length = 0;
    Rect.width = 0; /* 初始化矩形的长与宽 */
}

void Init_Point( void )
{
    Point.x = 10;
    Point.y = 10;   /* 初始化“点”的坐标 */
```

}

½6-21：如果多段代码重复做同一件事情，那么在函数的划分上可能存在问题。

说明：若此段代码各语句之间有实质性关联并且是完成同一件功能的，那么可考虑把此段代码构造成一个新的函数。

½6-22：功能不明确较小的函数，特别是仅有一个上级函数调用它时，应考虑把它合并到上级函数中，而不必单独存在。

说明：模块中函数划分的过多，一般会使函数间的接口变得复杂。所以过小的函数，特别是扇入很低的或功能不明确的函数，不值得单独存在。

½6-23：设计高扇入、合理扇出（小于7）的函数。

说明：扇出是指一个函数直接调用（控制）其它函数的数目，而扇入是指有多少上级函数调用它。

扇出过大，表明函数过分复杂，需要控制和协调过多的下级函数；而扇出过小，如总是 1，表明函数的调用层次可能过多，这样不利程序阅读和函数结构的分析，并且程序运行时会对系统资源如堆栈空间等造成压力。函数较合理的扇出（调度函数除外）通常是 3-5。扇出太大，一般是由于缺乏中间层次，可适当增加中间层次的函数。扇出太小，可把下级函数进一步分解多个函数，或合并到上级函数中。当然分解或合并函数时，不能改变要实现的功能，也不能违背函数间的独立性。

扇入越大，表明使用此函数的上级函数越多，这样的函数使用效率高，但不能违背函数间的独立性而单纯地追求高扇入。公共模块中的函数及底层函数应该有较高的扇入。

较良好的软件结构通常是顶层函数的扇出较高，中层函数的扇出较少，而底层函数则扇入到公共模块中。

½6-24：减少函数本身或函数间的递归调用。

说明：递归调用特别是函数间的递归调用（如 A->B->C->A），影响程序的可理解性；递归调用一般都占用较多的系统资源（如栈空间）；递归调用对程序的测试有一定影响。故除非为某些算法或功能的实现方便，应减少没必要的递归调用。

½6-25：仔细分析模块的功能及性能需求，并进一步细分，同时若有必要画出有关数据流图，据此来进行模块的函数划分与组织。

说明：函数的划分与组织是模块的实现过程中很关键的步骤，如何划分出合理的函数结构，关系到模块的最终效率和可维护性、可测性等。根据模块的功能图或/及数据流图映射出

函数结构是常用方法之一。

½6-26：改进模块中函数的结构，降低函数间的耦合度，并提高函数的独立性以及代码可读性、效率和可维护性。优化函数结构时，要遵守以下原则：

（1）不能影响模块功能的实现。

（2）仔细考查模块或函数出错处理及模块的性能要求并进行完善。

（3）通过分解或合并函数来改进软件结构。

（4）考查函数的规模，过大的要进行分解。

（5）降低函数间接口的复杂度。

（6）不同层次的函数调用要有较合理的扇入、扇出。

（7）函数功能应可预测。

（8）提高函数内聚。（单一功能的函数内聚最高）

说明：对初步划分后的函数结构应进行改进、优化，使之更为合理。

½6-27：在多任务操作系统的环境下编程，要注意函数可重入性的构造。

说明：可重入性是指函数可以被多个任务进程调用。在多任务操作系统中，函数是否具有可重入性是非常重要的，因为这是多个进程可以共用此函数的必要条件。另外，编译器是否提供可重入函数库，与它所服务的操作系统有关，只有操作系统是多任务时，编译器才有可能提供可重入函数库。如 DOS 下 BC 和 MSC 等就不具备可重入函数库，因为 DOS 是单用户单任务操作系统。

½6-28：避免使用BOOL参数。

说明：原因有二，其一是 BOOL 参数值无意义，TURE/FALSE 的含义是非常模糊的，在调用时很难知道该参数到底传达的是什么意思；其二是 BOOL 参数值不利于扩充。还有 NULL 也是一个无意义的单词。

½6-29： 对于提供了返回值的函数，在引用时最好使用其返回值。

½6-30：当一个过程（函数）中对较长变量（一般是结构的成员）有较多引用时，可以用一个意义相当的宏代替。

说明：这样可以增加编程效率和程序的可读性。

示例：在某过程中较多引用 TheReceiveBuffer[FirstSocket].byDataPtr，

则可以通过以下宏定义来代替：

```
# define pSOCKDATA TheReceiveBuffer[FirstScoket].byDataPtr
```

7 可测性

[1]7-1：在同一项目组或产品组内，要有一套统一的为集成测试与系统联调准备的调测开关及相应打印函数，并且要有详细的说明。

说明：本规则是针对项目组或产品组的。

[1]7-2：在同一项目组或产品组内，调测打印出的信息串的格式要有统一的形式。信息串中至少要有所在模块名（或源文件名）及行号。

说明：统一的调测信息格式便于集成测试。

[1]7-3：编程的同时要为单元测试选择恰当的测试点，并仔细构造测试代码、测试用例，同时给出明确的注释说明。测试代码部分应作为（模块中的）一个子模块，以方便测试代码在模块中的安装与拆卸（通过调测开关）。

说明：为单元测试而准备。

[1]7-4：在进行集成测试/系统联调之前，要构造好测试环境、测试项目及测试用例，同时仔细分析并优化测试用例，以提高测试效率。

说明：好的测试用例应尽可能模拟出程序所遇到的边界值、各种复杂环境及一些极端情况等。

[1]7-5：使用断言来发现软件问题，提高代码可测性。

说明：断言是对某种假设条件进行检查（可理解为若条件成立则无动作，否则应报告），它可以快速发现并定位软件问题，同时对系统错误进行自动报警。断言可以对在系统中隐藏很深，用其它手段极难发现的问题进行定位，从而缩短软件问题定位时间，提高系统的可测性。实际应用时，可根据具体情况灵活地设计断言。

示例：下面是 C 语言中的一个断言，用宏来设计的。（其中 NULL 为 0L）

```
#ifdef _EXAM_ASSERT_TEST_  // 若使用断言测试

void exam_assert( char * file_name, unsigned int line_no )
{
    printf( "\n[EXAM]Assert failed: %s, line %u\n",
            file_name, line_no );
    abort( );
```

```
}

#define  EXAM_ASSERT( condition )
    if (condition) // 若条件成立，则无动作
        NULL;
    else  // 否则报告
        exam_assert( __FILE__, __LINE__ )

#else  // 若不使用断言测试

#define EXAM_ASSERT(condition)  NULL

#endif  /* end of ASSERT */
```

¹**7-6：用断言来检查程序正常运行时不应发生但在调测时有可能发生的非法情况。**

¹**7-7：不能用断言来检查最终产品肯定会出现且必须处理的错误情况。**

说明：断言是用来处理不应该发生的错误情况的，对于可能会发生的且必须处理的情况要写防错程序，而不是断言。如某模块收到其它模块或链路上的消息后，要对消息的合理性进行检查，此过程为正常的错误检查，不能用断言来实现。

¹**7-8：对较复杂的断言加上明确的注释。**

说明：为复杂的断言加注释，可澄清断言含义并减少不必要的误用。

¹**7-9：用断言确认函数的参数。**

示例：假设某函数参数中有一个指针，那么使用指针前可对它检查，如下。

```
int exam_fun( unsigned char *str )
{
    EXAM_ASSERT( str != NULL );  // 用断言检查“假设指针不为空”这个条件

    ... //other program code
}
```

¹**7-10：用断言保证没有定义的特性或功能不被使用。**

示例：假设某通信模块在设计时，准备提供“无连接”和“连接” 这两种业务。但当前的版本中仅实现了“无连接”业务，且在此版本的正式发行版中，用户（上层模块）不应

产生“ 连接 ”业务的请求，那么在测试时可用断言检查用户是否使用“ 连接 ”业务。如下。

```
#define EXAM_CONNECTIONLESS 0 // 无连接业务
#define EXAM_CONNECTION     1 // 连接业务

int msg_process( EXAM_MESSAGE *msg )
{
   unsigned char service; /* message service class */

   EXAM_ASSERT( msg != NULL );

service = get_msg_service_class( msg );

   EXAM_ASSERT( service != EXAM_CONNECTION ); // 假设不使用连接业务

   ...  //other program code
}
```

¹7-11：用断言对程序开发环境（OS/Compiler/Hardware）的假设进行检查。

说明：程序运行时所需的软硬件环境及配置要求，不能用断言来检查，而必须由一段专门代码处理。用断言仅可对程序开发环境中的假设及所配置的某版本软硬件是否具有某种功能的假设进行检查。如某网卡是否在系统运行环境中配置了，应由程序中正式代码来检查；而此网卡是否具有某设想的功能，则可由断言来检查。

对编译器提供的功能及特性假设可用断言检查，原因是软件最终产品（即运行代码或机器码）与编译器已没有任何直接关系，即软件运行过程中（注意不是编译过程中）不会也不应该对编译器的功能提出任何需求。

示例：用断言检查编译器的 int 型数据占用的内存空间是否为 2，如下。

```
EXAM_ASSERT( sizeof( int ) == 2 );
```

¹7-12：正式软件产品中应把断言及其它调测代码去掉（即把有关的调测开关关掉）。

说明：加快软件运行速度。

¹7-13：在软件系统中设置与取消有关测试手段，不能对软件实现的功能等产生影响。

说明：即有测试代码的软件和关掉测试代码的软件，在功能行为上应一致。

¹7-14：用调测开关来切换软件的DEBUG版和正式版，而不要同时存在正式版本和DEBUG版本

的不同源文件，以减少维护的难度。

¹7-15：软件的DEBUG版本和发行版本应该统一维护，不允许分家，并且要时刻注意保证两个版本在实现功能上的一致性。

½7-1：在编写代码之前，应预先设计好程序调试与测试的方法和手段，并设计好各种调测开关及相应测试代码如打印函数等。

说明：程序的调试与测试是软件生存周期中很重要的一个阶段，如何对软件进行较全面、高率的测试并尽可能地找出软件中的错误就成为很关键的问题。因此在编写源代码之前，除了要有一套比较完善的测试计划外，还应设计出一系列代码测试手段，为单元测试、集成测试及系统联调提供方便。

½7-2：调测开关应分为不同级别和类型。

说明：调测开关的设置及分类应从以下几方面考虑：针对模块或系统某部分代码的调测；针对模块或系统某功能的调测；出于某种其它目的，如对性能、容量等的测试。这样做便于软件功能的调测，并且便于模块的单元测试、系统联调等。

½7-3：编写防错程序，然后在处理错误之后可用断言宣布发生错误。

示例：假如某模块收到通信链路上的消息，则应对消息的合法性进行检查，若消息类别不是通信协议中规定的，则应进行出错处理，之后可用断言报告，如下例。

```
#ifdef _EXAM_ASSERT_TEST_ // 若使用断言测试

/* Notice: this function does not call 'abort' to exit program */
void assert_report( char * file_name, unsigned int line_no )
{
   printf( "\n[EXAM]Error Report: %s, line %u\n",
          file_name, line_no );
}

#define  ASSERT_REPORT( condition )
   if ( condition ) // 若条件成立，则无动作
      NULL;
   else // 否则报告
      assert_report ( __FILE__, __LINE__ )
```

```
#else // 若不使用断言测试

#define ASSERT_REPORT( condition )  NULL

#endif /* end of ASSERT */

int msg_handle( unsigned char msg_name, unsigned char * msg )
{
    switch( msg_name )
    {
        case MSG_ONE:
            ... // 消息 MSG_ONE 处理
            return MSG_HANDLE_SUCCESS;

            ... // 其它合法消息处理

        default:
            ... // 消息出错处理
            ASSERT_REPORT( FALSE );  // “合法”消息不成立，报告
            return MSG_HANDLE_ERROR;
    }
}
```

8 程序效率

¹8-1：编程时要经常注意代码的效率。

说明：代码效率分为全局效率、局部效率、时间效率及空间效率。全局效率是站在整个系统的角度上的系统效率；局部效率是站在模块或函数角度上的效率；时间效率是程序处理输入任务所需的时间长短；空间效率是程序所需内存空间，如机器代码空间大小、数据空间大小、栈空间大小等。

¹8-2：在保证软件系统的正确性、稳定性、可读性及可测性的前提下，提高代码效率。

说明：不能一味地追求代码效率，而对软件的正确性、稳定性、可读性及可测性造成影响。

¹8-3：局部效率应为全局效率服务，不能因为提高局部效率而对全局效率造成影响。

¹8-4：通过对系统数据结构的划分与组织的改进，以及对程序算法的优化来提高空间效率。

说明：这种方式是解决软件空间效率的根本办法。

示例：如下记录学生学习成绩的结构不合理。

```
typedef unsigned char  BYTE;
typedef unsigned short WORD;

typedef struct STUDENT_SCORE_STRU


    BYTE name[8];
    BYTE age;
    BYTE sex;
    BYTE class;
    BYTE subject;
    float score;
} STUDENT_SCORE;
```

因为每位学生都有多科学习成绩，故如上结构将占用较大空间。应如下改进（分为两个结构），总的存贮空间将变小，操作也变得更方便。

```
typedef struct STUDENT_STRU
{
```

```
    BYTE name[8];
    BYTE age;
    BYTE sex;
    BYTE class;
} STUDENT;

typedef struct STUDENT_SCORE_STRU
{
    WORD student_index;
    BYTE subject;
    float score;
} STUDENT_SCORE;
```

¹8-5：循环体内工作量最小化。

说明：应仔细考虑循环体内的语句是否可以放在循环体之外，使循环体内工作量最小，从而提高程序的时间效率。

示例：如下代码效率不高。

```
for (ind = 0; ind < MAX_ADD_NUMBER; ind++)
{
    sum += ind;
    back_sum = sum; /* backup sum */
}
```

语句“back_sum = sum;”完全可以放在 for 语句之后，如下。

```
for (ind = 0; ind < MAX_ADD_NUMBER; ind++)
{
    sum += ind;
}
back_sum  = sum; /* backup sum */
```

½8-1：仔细分析有关算法，并进行优化。

½8-2：仔细考查、分析系统及模块处理输入（如事务、消息等）的方式，并加以改进。

½8-3：对模块中函数的划分及组织方式进行分析、优化，改进模块中函数的组织结构，提高程序效率。

说明：软件系统的效率主要与算法、处理任务方式、系统功能及函数结构有很大关系，仅在代码上下功夫一般不能解决根本问题。

½**8-4**：*编程时，要随时留心代码效率；优化代码时，要考虑周全。*

½**8-5**：*不应花过多的时间拼命地提高调用不很频繁的函数代码效率。*

说明：对代码优化可提高效率，但若考虑不周很有可能引起严重后果。

½**8-6**：*要仔细地构造或直接用汇编编写调用频繁或性能要求极高的函数。*

说明：只有对编译系统产生机器码的方式以及硬件系统较为熟悉时，才可使用汇编嵌入方式。嵌入汇编可提高时间及空间效率，但也存在一定风险。

½**8-7**：*在保证程序质量的前提下，通过压缩代码量、去掉不必要代码以及减少不必要的局部和全局变量，来提高空间效率。*

说明：这种方式对提高空间效率可起到一定作用，但往往不能解决根本问题。

½**8-8**：*在多重循环中，应将最忙的循环放在最内层。*

说明：减少 CPU 切入循环层的次数。

示例：如下代码效率不高。

```
for (row = 0; row < 100; row++)
{
    for (col = 0; col < 5; col++)
    {
        sum += a[row][col];
    }
}
```

可以改为如下方式，以提高效率。

```
for (col = 0; col < 5; col++)
{
    for (row = 0; row < 100; row++)
    {
        sum += a[row][col];
    }
}
```

½8-9：尽量减少循环嵌套层次。

½8-10：避免循环体内含判断语句，应将循环语句置于判断语句的代码块之中。

说明：目的是减少判断次数。循环体中的判断语句是否可以移到循环体外，要视程序的具体情况而言，一般情况，与循环变量无关的判断语句可以移到循环体外，而有关的则不可以。

示例：如下代码效率稍低。

```
for (ind = 0; ind < MAX_RECT_NUMBER; ind++)
{
    if (data_type == RECT_AREA)
    {
        area_sum += rect_area[ind];
    }
    else
    {
        rect_length_sum += rect[ind].length;
        rect_width_sum += rect[ind].width;
    }
}
```

因为判断语句与循环变量无关，故可如下改进，以减少判断次数。

```
if (data_type == RECT_AREA)
{
    for (ind = 0; ind < MAX_RECT_NUMBER; ind++)
    {
        area_sum += rect_area[ind];
    }
}
else
{
    for (ind = 0; ind < MAX_RECT_NUMBER; ind++)
    {
        rect_length_sum += rect[ind].length;
        rect_width_sum  += rect[ind].width;
    }
```

```
}
```

½8-11：尽量用乘法或其它方法代替除法，特别是浮点运算中的除法。

说明：浮点运算除法要占用较多 CPU 资源。

示例：如下表达式运算可能要占较多 CPU 资源。

```
#define PAI 3.1416
radius = circle_length / (2 * PAI);
```

应如下把浮点除法改为浮点乘法。

```
#define PAI_RECIPROCAL (1 / 3.1416 ) // 编译器编译时，将生成具体浮点数
radius = circle_length * PAI_RECIPROCAL / 2;
```

½8-12：不要一味追求紧凑的代码。

说明：因为紧凑的代码并不代表高效的机器码。

9 质量保证

[1]9-1：在软件设计过程中构筑软件质量。

[1]9-2：代码质量保证优先原则

（1）正确性，指程序要实现设计要求的功能。

（2）稳定性、安全性，指程序稳定、可靠、安全。

（3）可测试性，指程序要具有良好的可测试性。

（4）规范/可读性，指程序书写风格、命名规则等要符合规范。

（5）全局效率，指软件系统的整体效率。

（6）局部效率，指某个模块/子模块/函数的本身效率。

（7）个人表达方式/个人方便性，指个人编程习惯。

[1]9-3：只引用属于自己的存贮空间。

说明：若模块封装的较好，那么一般不会发生非法引用他人的空间。

[1]9-4：防止引用已经释放的内存空间。

说明：在实际编程过程中，稍不留心就会出现在一个模块中释放了某个内存块（如 C 语言指针），而另一模块在随后的某个时刻又使用了它。要防止这种情况发生。

[1]9-5：过程/函数中分配的内存，在过程/函数退出之前要释放。

[1]9-6：过程/函数中申请的（为打开文件而使用的）文件句柄，在过程/函数退出之前要关闭。

说明：分配的内存不释放以及文件句柄不关闭，是较常见的错误，而且稍不注意就有可能发生。这类错误往往会引起很严重后果，且难以定位。

示例：下函数在退出之前，没有把分配的内存释放。

```
typedef unsigned char BYTE;

int example_fun( BYTE gt_len, BYTE *gt_code )
{
    BYTE *gt_buf;

    gt_buf = (BYTE *) malloc (MAX_GT_LENGTH);
    ...  //program code, include check gt_buf if or not NULL.
```

```
    /* global title length error */
    if (gt_len > MAX_GT_LENGTH)
    {
        return GT_LENGTH_ERROR; // 忘了释放 gt_buf
    }

    ... // other program code
}
```

应改为如下。

```
int example_fun( BYTE gt_len, BYTE *gt_code )
{
    BYTE *gt_buf;

    gt_buf = (BYTE * ) malloc ( MAX_GT_LENGTH );
    ... // program code, include check gt_buf if or not NULL.

    /* global title length error */
    if (gt_len > MAX_GT_LENGTH)
    {
        free( gt_buf ); // 退出之前释放 gt_buf
        return GT_LENGTH_ERROR;
    }

    ... // other program code
}
```

¹9-7：防止内存操作越界。

说明：内存操作主要是指对数组、指针、内存地址等的操作。内存操作越界是软件系统主要错误之一，后果往往非常严重，所以当我们进行这些操作时一定要仔细小心。

示例：假设某软件系统最多可由 10 个用户同时使用，用户号为 1-10，那么如下程序存在问题。

```
#define MAX_USR_NUM 10
unsigned char usr_login_flg[MAX_USR_NUM]= "";
```

```
void set_usr_login_flg( unsigned char usr_no )
{
    if (!usr_login_flg[usr_no])
    {
        usr_login_flg[usr_no]= TRUE;
    }
}
```

当 usr_no 为 10 时，将使用 usr_login_flg 越界。可采用如下方式解决。

```
void set_usr_login_flg( unsigned char usr_no )
{
    if (!usr_login_flg[usr_no - 1])
    {
        usr_login_flg[usr_no - 1]= TRUE;
    }
}
```

¹9-8：认真处理程序所能遇到的各种出错情况。

¹9-9：系统运行之初，要初始化有关变量及运行环境，防止未经初始化的变量被引用。

¹9-10：系统运行之初，要对加载到系统中的数据进行一致性检查。

说明：使用不一致的数据，容易使系统进入混乱状态和不可知状态。

¹9-11：严禁随意更改其它模块或系统的有关设置和配置。

说明：编程时，不能随心所欲地更改不属于自己模块的有关设置如常量、数组的大小等。

¹9-12：不能随意改变与其它模块的接口。

¹9-13：充分了解系统的接口之后，再使用系统提供的功能。

示例：在 B 型机的各模块与操作系统的接口函数中，有一个要由各模块负责编写的初始化过程，此过程在软件系统加载完成后，由操作系统发送的初始化消息来调度。因此就涉及到初始化消息的类型与消息发送的顺序问题，特别是消息顺序，若没搞清楚就开始编程，很容易引起严重后果。以下示例引自 B 型曾出现过的实际代码，其中使用了 FID_FETCH_DATA 与 FID_INITIAL 初始化消息类型，注意 B 型机的系统是在

FID_FETCH_DATA 之前发送 FID_INITIAL 的。

```
MID alarm_module_list[MAX_ALARM_MID];

int FAR SYS_ALARM_proc( FID function_id, int handle )
{
    _UI i, j;

    switch ( function_id )
    {
        ... // program code

        case FID_INITAIL:
            for (i = 0; i < MAX_ALARM_MID; i++)
            {
                if (alarm_module_list[i]== BAM_MODULE // **)
                   || (alarm_module_list[i]== LOCAL_MODULE)
                {

                    for (j = 0; j < ALARM_CLASS_SUM; j++)
                    {
                        FAR_MALLOC( ... );
                    }
                }
            }

            ... // program code

            break;

        case FID_FETCH_DATA:

            ... // program code

            Get_Alarm_Module( );  // 初始化 alarm_module_list
```

```
            break;

        ... // program code
    }
}
```

由于FID_INITIAL是在FID_FETCH_DATA之前执行的，而初始化alarm_module_list是在FID_FETCH_DATA中进行的，故在FID_INITIAL中（**）处引用alarm_module_list变量时，它还没有被初始化。这是个严重错误。

应如下改正：要么把Get_Alarm_Module函数放在FID_INITIAL中（**）之前；要么就必须考虑（**）处的判断语句是否可以用（不使用alarm_module_list变量的）其它方式替代，或者是否可以取消此判断语句。

¹9-14：编程时，要防止差1错误。

说明：此类错误一般是由于把“<=”误写成“<”或“>=”误写成“>”等造成的，由此引起的后果，很多情况下是很严重的，所以编程时，一定要在这些地方小心。当编完程序后，应对这些操作符进行彻底检查。

¹9-15：要时刻注意易混淆的操作符。当编完程序后，应从头至尾检查一遍这些操作符，以防止拼写错误。

说明：形式相近的操作符最容易引起误用，如C/C++中的“=”与“==”、“|”与“||”、“&”与“&&”等，若拼写错了，编译器不一定能够检查出来。

示例：如把“&”写成“&&”，或反之。

```
ret_flg = (pmsg->ret_flg & RETURN_MASK);
```

被写为：

```
ret_flg = (pmsg->ret_flg && RETURN_MASK);

rpt_flg = (VALID_TASK_NO( taskno ) && DATA_NOT_ZERO( stat_data ));
```

被写为：

```
rpt_flg = (VALID_TASK_NO( taskno ) & DATA_NOT_ZERO( stat_data ));
```

¹9-16：有可能的话，if语句尽量加上else分支，对没有else分支的语句要小心对待；switch语句必须有default分支。

¹9-17：Unix下，多线程的中的子线程退出必需采用主动退出方式，即子线程应return出口。

¹9-18：不要滥用goto语句。

说明：goto 语句会破坏程序的结构性，所以除非确实需要，最好不使用 goto 语句。

½9-1：不使用与硬件或操作系统关系很大的语句，而使用建议的标准语句，以提高软件的可移植性和可重用性。

½9-2：除非为了满足特殊需求，避免使用嵌入式汇编。

说明：程序中嵌入式汇编，一般都对可移植性有较大的影响。

½9-3：精心地构造、划分子模块，并按"接口"部分及"内核"部分合理地组织子模块，以提高"内核"部分的可移植性和可重用性。

说明：对不同产品中的某个功能相同的模块，若能做到其内核部分完全或基本一致，那么无论对产品的测试、维护，还是对以后产品的升级都会有很大帮助。

½9-4：精心构造算法，并对其性能、效率进行测试。

½9-5：对较关键的算法最好使用其它算法来确认。

½9-6：时刻注意表达式是否会上溢、下溢。

示例：如下程序将造成变量下溢。

```
unsigned char size ;
while (size-- >= 0) // 将出现下溢
{
    ... // program code
}
```

当 size 等于 0 时，再减 1 不会小于 0，而是 0xFF，故程序是一个死循环。应如下修改。

```
char size; // 从 unsigned char 改为 char
while (size-- >= 0)
{
    ... // program code
}
```

½9-7：使用变量时要注意其边界值的情况。

示例：如C语言中字符型变量，有效值范围为-128到127。故以下表达式的计算存在一定风险。

```
char chr = 127;
int sum = 200;

chr += 1; // 127为chr的边界值，再加1将使chr上溢到-128，而不是128。
sum += chr; // 故sum的结果不是328，而是72。
```

若chr与sum为同一种类型，或表达式按如下方式书写，可能会好些。

```
sum = sum + chr + 1;
```

½**9-8**：*留心程序机器码大小（如指令空间大小、数据空间大小、堆栈空间大小等）是否超出系统有关限制。*

½**9-9**：*为用户提供良好的接口界面，使用户能较充分地了解系统内部运行状态及有关系统出错情况。*

½**9-10**：*系统应具有一定的容错能力，对一些错误事件（如用户误操作等）能进行自动补救。*

½**9-11**：*对一些具有危险性的操作代码（如写硬盘、删数据等）要仔细考虑，防止对数据、硬件等的安全构成危害，以提高系统的安全性。*

½**9-12**：*使用第三方提供的软件开发工具包或控件时，要注意以下几点：*

（1）充分了解应用接口、使用环境及使用时注意事项。

（2）不能过分相信其正确性。

（3）除非必要，不要使用不熟悉的第三方工具包与控件。

说明：使用工具包与控件，可加快程序开发速度，节省时间，但使用之前一定对它有较充分的了解，同时第三方工具包与控件也有可能存在问题。

½**9-13**：*资源文件（多语言版本支持），如果资源是对语言敏感的，应让该资源与源代码文件脱离，具体方法有下面几种：使用单独的资源文件、DLL文件或其它单独的描述文件（如数据库格式）*

10 代码编辑、编译、审查

¹10-1：打开编译器的所有告警开关对程序进行编译。

¹10-2：在产品软件（项目组）中，要统一编译开关选项。

¹10-3：通过代码走读及审查方式对代码进行检查。

说明：代码走读主要是对程序的编程风格如注释、命名等以及编程时易出错的内容进行检查，可由开发人员自己或开发人员交叉的方式进行；代码审查主要是对程序实现的功能及程序的稳定性、安全性、可靠性等进行检查及评审，可通过自审、交叉审核或指定部门抽查等方式进行。

¹10-4：测试部测试产品之前，应对代码进行抽查及评审。

½10-1：编写代码时要注意随时保存，并定期备份，防止由于断电、硬盘损坏等原因造成代码丢失。

½10-2：同产品软件（项目组）内，最好使用相同的编辑器，并使用相同的设置选项。

说明：同一项目组最好采用相同的智能语言编辑器，如 Muiti Editor，Visual Editor 等，并设计、使用一套缩进宏及注释宏等，将缩进等问题交由编辑器处理。

½10-3：要小心地使用编辑器提供的块拷贝功能编程。

说明：当某段代码与另一段代码的处理功能相似时，许多开发人员都用编辑器提供的块拷贝功能来完成这段代码的编写。由于程序功能相近，故所使用的变量、采用的表达式等在功能及命名上可能都很相近，所以使用块拷贝时要注意，除了修改相应的程序外，一定要把使用的每个变量仔细查看一遍，以改成正确的。不应指望编译器能查出所有这种错误，比如当使用的是全局变量时，就有可能使某种错误隐藏下来。

½10-4：合理地设计软件系统目录，方便开发人员使用。

说明：方便、合理的软件系统目录，可提高工作效率。目录构造的原则是方便有关源程序的存储、查询、编译、链接等工作，同时目录中还应具有工作目录----所有的编译、链接等工作应在此目录中进行，工具目录----有关文件编辑器、文件查找等工具可存放在此目录中。

½10-5：某些语句经编译后产生告警，但如果你认为它是正确的，那么应通过某种手段去掉告警信息。

说明：在 Borland C/C++中，可用“#pragma warn”来关掉或打开某些告警。

示例：

```
#pragma warn -rvl // 关闭告警
int examples_fun( void )
{
    // 程序，但无 return 语句。
}
#pragma warn +rvl // 打开告警
```

编译函数 examples_fun 时本应产生“函数应有返回值”告警，但由于关掉了此告警信息显示，所以编译时将不会产生此告警提示。

½10-6：使用代码检查工具（如C语言用PC-Lint）对源程序检查。

½10-7：使用软件工具（如 LogiSCOPE）进行代码审查。

11 代码测试、维护

¹11-1：单元测试要求至少达到语句覆盖。

¹11-2：单元测试开始要跟踪每一条语句，并观察数据流及变量的变化。

¹11-3：清理、整理或优化后的代码要经过审查及测试。

¹11-4：代码版本升级要经过严格测试。

¹11-5：使用工具软件对代码版本进行维护。

¹11-6：正式版本上软件的任何修改都应有详细的文档记录。

*½**11-1**：发现错误立即修改，并且要记录下来。*

*½**11-2**：关键的代码在汇编级跟踪。*

*½**11-3**：仔细设计并分析测试用例，使测试用例覆盖尽可能多的情况，以提高测试用例的效率。*

*½**11-4**：尽可能模拟出程序的各种出错情况，对出错处理代码进行充分的测试。*

*½**11-5**：仔细测试代码处理数据、变量的边界情况。*

*½**11-6**：保留测试信息，以便分析、总结经验及进行更充分的测试。*

*½**11-7**：不应通过“试”来解决问题，应寻找问题的根本原因。*

*½**11-8**：对自动消失的错误进行分析，搞清楚错误是如何消失的。*

*½**11-9**：修改错误不仅要治表，更要治本。*

*½**11-10**：测试时应设法使很少发生的事件经常发生。*

*½**11-11**：明确模块或函数处理哪些事件，并使它们经常发生。*

*½**11-12**： 坚持在编码阶段就对代码进行彻底的单元测试，不要等以后的测试工作来发现问题。*

½11-13：去除代码运行的随机性（如去掉无用的数据、代码及尽可能防止并注意函数中的“内部寄存器”等），让函数运行的结果可预测，并使出现的错误可再现。

12 宏

[1]12-1：用宏定义表达式时，要使用完备的括号。

示例：如下定义的宏都存在一定的风险。

```
#define RECTANGLE_AREA( a, b ) a * b
#define RECTANGLE_AREA( a, b ) (a * b)
#define RECTANGLE_AREA( a, b ) (a) * (b)
```

正确的定义应为：

```
#define RECTANGLE_AREA( a, b ) ((a) * (b))
```

[1]12-2:将宏所定义的多条表达式放在大括号中。

示例：下面的语句只有宏的第一条表达式被执行。为了说明问题，for 语句的书写稍不符规范。

```
#define INTI_RECT_VALUE( a, b )\
   a = 0;\
   b = 0;

for (index = 0; index < RECT_TOTAL_NUM; index++)
   INTI_RECT_VALUE( rect.a, rect.b );
```

正确的用法应为：

```
#define INTI_RECT_VALUE( a, b )\
{\
   a = 0;\
   b = 0;\
}

for (index = 0; index < RECT_TOTAL_NUM; index++)
{
```

```
  INTI_RECT_VALUE( rect[index].a, rect[index].b );
}
```

[1]**12-3**:**使用宏时，不允许参数发生变化。**

示例：如下用法可能导致错误。

```
#define SQUARE( a ) ((a) * (a))

int a = 5;
int b;
b = SQUARE( a++ ); // 结果：a = 7，即执行了两次增1。
```

正确的用法是：

```
b = SQUARE( a );
a++; // 结果：a = 6，即只执行了一次增1。
```